A COMPARISON OF DEEP LEARNING AND CONVENTIONAL ALGORITHMS IN NARROW SPACE NAVIGATION

By

MINGXIN LIU

Dr. Wyatt S. Newman

Thesis Advisor

Submitted in partial fulfillment of the requirements for the degree of Master of Science

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

August 2020

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

Mingxin Liu

Candidate for the degrees of Master of Science

Committee Chair

Wyatt Newman

Committee Member

Gregory Lee

Committee Member

M. Cenk Cavusoglu

Date of Defense

June 12th 2020

*****We also certify that written approval has been obtained

for any proprietary material contained therein.

Table of Contents

List of Figures

A Comparison Of Deep Learning And Conventional Algorithms in Narrow Space Navigation

Abstract

By

MINGXIN LIU

This research explores the implementation of mobile-robot navigation using a neural network. A wall-following (bug) algorithm is implemented both conventionally and through network training. Using simulations in Gazebo, extensive labelled, virtual experience is obtained. This virtual experience is used to train networks that interpret 3-D point-cloud data to predict the consequences of controller options. The neural-net implementation is compared and contrasted with a more conventional, algorithmic implementation of a corresponding bug algorithm. Results indicate that the neural-net training approach is effective for robot navigation, and suggests advantages over explicit implementation.

1 Introduction

The recent resurgence of neural networks, with the successes of "deep learning", motivates reexamining algorithmic approaches to intelligent behaviors. One area of great interest is autonomous navigation, including autonomous vehicles on roads, off-road navigation, and navigation of mobile robots within buildings. The present work was motivated by earlier successes, by T.J. Pech, in training a simulated mobile robot to navigate intelligently in challenging terrain [24]. In this prior work, a robot navigated through complex environments while avoiding endangering the robot, e.g. through collisions, tipping, or falling off cliffs. This prior work showed that a neural-net system could learn from simulated successes and mistakes and embed that experience as part of making good choices. A limitation of the prior research is that the robot would successfully "wander," but it did not purposefully navigate to a specified goal.

The present research extends this prior effort by introducing neural-net based navigation that is goal oriented. Specifically, a "bug" algorithm [41] is explored for neural-net implementation. In contrast to [24], the present research assumes a simpler terrain (indoor navigation on smooth floors), but it introduces more challenging passageways and the pursuit of specific goals.

The use of a neural network for implementing navigation may seem an odd fit for a problem that has been treated with algorithms such as tree searching, wall following, visibility graphs, and other conventional, hard-coded approaches. A motivation for a neural-net approach is the deceptive complexity of implementing a "bug" algorithm.

Heuristically, the bug algorithm seems simple; essentially, follow a wall (and a virtual wall representing a default straight-line path to the goal). If a child is asked to draw a wall-following path, with only meager instruction, one would typically get the desired result: a tracing of the boundary of obstacles. This thus seems like a policy that should be simple to implement as an algorithm. In practice, however, one finds that such implementation is not as well defined as it would seem.

One typically approaches this problem by representing the perception of the environment as a configuration-space (C-space) equivalent, to account for the diameter of the robot. Subsequently, planning consists of motion of a dimensionless point within a 2-D map. However, if the robot is not circular, the "diameter" of the robot is not defined. If one chooses the diameter of an enclosing circle, this conservative approximation can result in precluding the robot from essential motions, such as passing through a doorway.

The specific robot considered in the present work is based on a mobile robot at Case Western Reserve University. This robot has a rectangular outline. It can navigate through doorways, but a bounding circle for this robot would not be capable of passing through a doorway. Similarly, obstacles in hallways (including people) should be manageable, but not with the conservative circular-robot approximation.

Since orientation of the robot matters, a more formal planning approach would be to consider orientation as a third dimension and construct a 3-D C-space. However, the bug algorithm, which traces boundaries of 2-D regions, is undefined for 3-D maps.

Nonetheless, if a volunteer were asked to drive the robot with a joystick to execute a wall-following algorithm, an experienced driver would be able to execute the intent without

further instructions, without collisions, without getting stuck, and ultimately with successful goal attainment (for attainable goals). This thought experiment suggests that implementation of a bug algorithm is intuitive, even though it is not mathematically well defined for non-circular mobile robots. This thesis investigates such implementation with a neural network.

The present research builds on a key notion of Pech's research [24]: that intelligent navigation can be achieved through virtual experience. Current autonomous vehicle developers are acquiring network training data from recordings of actual vehicle navigation (whether autonomous or by a human driver). However, such experience is lacking in "bad" cases--e.g. involving collisions. In contrast, simulations of vehicle driving with a realistic physics model offer the ability to experience virtual collisions. By this means, one can build up virtual experiences that are both bad and good, which is essential for neural-net training.

This thesis presents implementations and results of a wall-following "bug" algorithm, comparing and contrasting an algorithmic implementation vs. a neural-net implementation. The algorithms are evaluated in simulation using Gazebo. The simulation includes modeling the environment, the vehicle controls, the vehicle sensors, and the vehicle dynamics. The robot model was based on a physical mobile robot at CWRU and its sensors. The environment models included hallways in a CWRU building. The implementation, using ROS and TensorFlow, was built to be immediately portable from the simulated robot to the physical robot.

1.1 System Overview

ROS (Robot Operating System) is the most current standard programming platform for modern robots. The robot operating system was initiated at the Stanford University Artificial

Intelligence Lab as early as 2007, and now ROS is widely used around the world in academia, government and industry [25]. ROS provides a framework to design software that is distributed among asynchronous "nodes" and which is easily implemented across a network of computers. A key foundation of ROS is its means for communication among nodes, which includes both a publish-subscribe mechanism and a client/service mechanism. ROS also provides extensive tools and libraries and helps with debugging, visualizing, logging and testing [25]. Sensor interfacing, sensory interpretation, motion control, planning, simulation and human interfaces can all be programmed using ROS nodes. The nodes are usually written in either Python or C++ executing individually and asynchronously. Nodes can provide services for other nodes and publish messages on topics for sensory input, sensory interpretation, planning, and motion control [25]. The tools ROS has for simulation and visualization are: Gazebo and RViz [24]. Gazebo is a 3D simulator that allows users to generate models such as robots and obstacles. Gazebo runs a physics engine that simulates illumination, gravity, collision and other physics properties to emulate the physical world. With Gazebo, much testing that would be costly and dangerous in hardware can be carried out with zero cost and liability in simulation [27]. Hence, low level testing in simulation could show a proof of concept before physical testing.

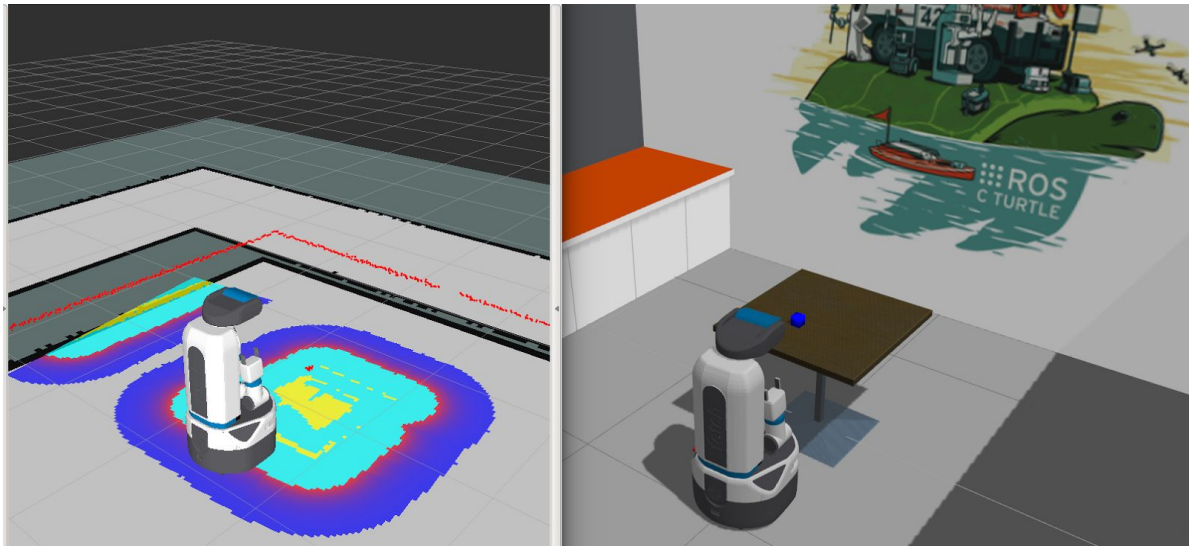


Figure 1.1 Fetch view of Gazebo(right) and RViz(left) [39]

RViz is a ROS visualizing interface for sensor output. Figure 1.1 shows a combined image of RViz and Gazebo visualization. RViz publishes the sensor outputs from the Gazebo simulation. In other words, once a sensor is created in Gazebo, the sensor output can be published as "message" on a "topic" through ROS. ROS nodes can subscribe to such topics and receive the published messages. Rviz is able to display sensor information by subscribing to such topics.

1.2 The advantage of simulation

The primary purpose of simulation is to provide instant feedback. Gazebo simulation has a powerful interface that has visual and analytical data. The simulation testing can be easily designed and modified to fulfill the needs of a research topic. It is an excellent tool for proof of concept. Candidate algorithms, which might only approximately correspond to mathematical models and assumptions, may be carried out in Gazebo to test the viability of an idea before pursuing it further.

Another primary advantage of simulation is its cheap implementation. Gazebo and ROS are both free to download and utilize. Furthermore, for research that involves collision and risky experiments, running tests in simulation is far cheaper and easier. In particular, acquiring training data for neural networks using physical hardware may be impractical, due to both time and risk--particularly if training examples must include examples of "bad" behavior. In contrast, a large array of experiences can be obtained automatically through simulation, including both safe and catastrophic examples. Specifically, collecting training data for a mobile robot is challenging if the intent of the training data is to predict collisions. It would be necessary to acquire enough examples of actual collisions to be included in the training set. With Gazebo and Rviz, such explorations are practical [32].

The results generated from the simulation alone, however, do at some level vary from the ones generated with physical sensors and physical dynamics. But for a proof of concept, the simulation is still very helpful. It is also noteworthy that simulation provides a simple means to debug and illustrate results. The results obtained from simulation can be visually demonstrated to peers and students who are not technically trained with an engineering background.

The present research builds on and benefits from the prior work of [32], including many of the concepts and some of the actual code. Building on prior efforts enabled rapid progress.

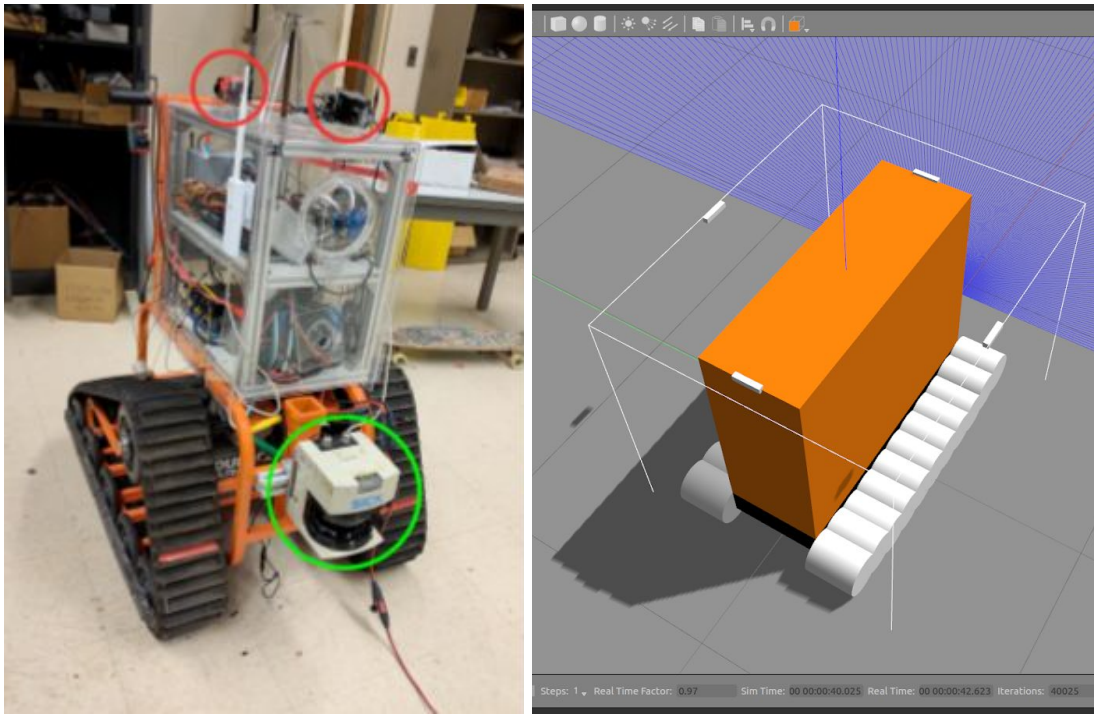


Figure 1.2 Physical robot(left) and simulated model counterpart (right) [24]

Figure 1.2 shows the physical robot in the Glennen lab on the left and the simulation counterpart of it on the right. The simulated robot assumes four Kinect sensors as well as a 3D Light Detection and Ranging (LIDAR) sensor.

Precursor research:

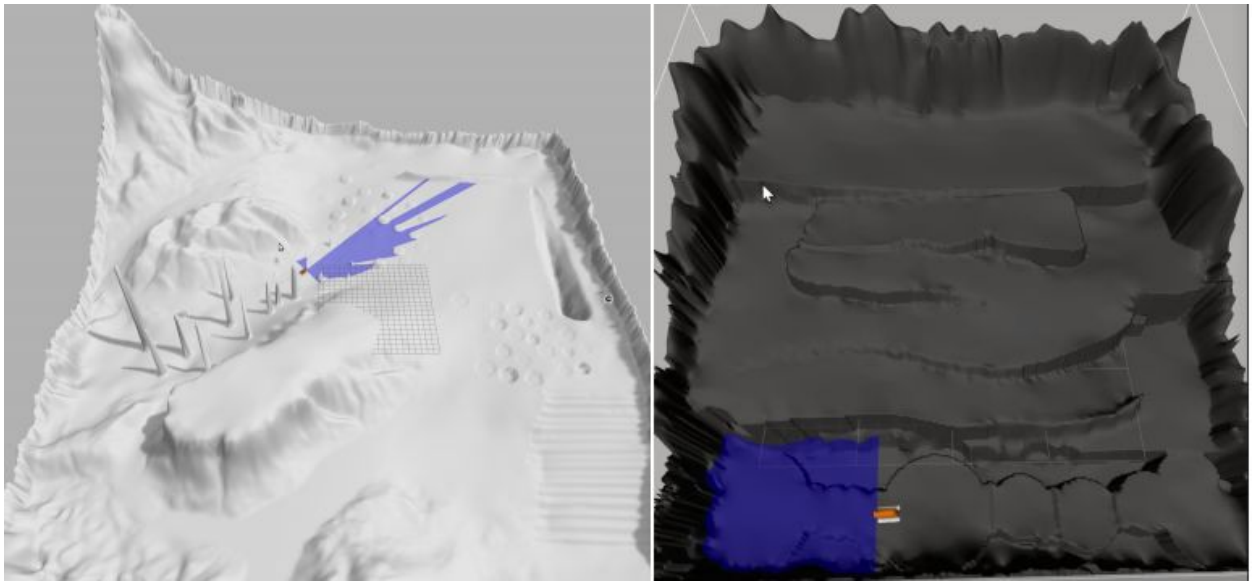


Figure 1.3 Terrain maps from Pech's research [24]

Pech's prior research [24 Pech] focused mainly on outdoor navigation and navigation strategies for driving on various terrain conditions such as bumps, slopes, hills and edges. A neural network was trained to predict failure modes such as tilting, colliding and flipping over. While his results were promising, physical validation would be difficult to accomplish, as it would require testing in odd environments as well as incurring substantial risk to the robot.

The present research focuses on indoor navigation. Indoor scenarios are easy to test, and the hazards are easier for a human to oversee and protect against, making experimental testing less risky. The present research also extends Pech's research to incorporate intelligent navigation, rather than simple wandering.

1.3 Creating Models and Maps in Gazebo

The Gazebo interface can be launched with the command line 'roslaunch gazebo_ros empty_world.launch'. This command launches the Gazebo interface with only a ground plane in the world. Models can be launched from a pre-existing Gazebo models library, or can be created through 3D model generation software such as Meshlab. Models can also be recorded with a 3D scanner and be created with the generated STL file from the scanner.

The components required to describe a model are: Link, Collision, Visual, Inertial, Sensor, Light, Joint and Plugin [29]. A link is the physical property that connects a physical model to another physical model. This link could exist on a physical joint, but it could also be two links without any joints. An object without a link does not possess any physical properties in simulation, so the link is the most important property that defines the object in simulations. "Collision" is the element that defines a surface geometry that can be used to compute collision checking. This is determined by its sub-elements of contact coefficients: Mu(friction coefficient), Kp(Stiffness coefficient) and Kd(Damping coefficient)[30]. The "visual" specifies how to render a link graphically.. "Inertial" is the element that gives the link dynamic properties, including mass, rotational inertia, and friction. Joints define child/parent relations between links, often with an actuator that allows motion between the connected links [29]. A sensor is a device that collects data (e.g. camera data, LIDAR, IMU, joint angles, joint velocities, force/torque, etc.) and outputs the data via messages on a ROS topic. While sensors are not needed to perform dynamic simulations, they are essential for implementing perception and intelligent behaviors.

Building a model has multiple steps. First, a real world model can be scanned and exported as an STL or OBJ file and this file can be launched in Gazebo after scaling to the right size. Other than exporting data from a real model, the models can also be created with Blender [31]. The traffic cones for example were created from the real cones in the Glennen lab. Cones were brought to the ThinkBox located in the veale center at CWRU and scanned with the 3D scan arm. The 3D scanned model was generated with Autodesk Fusion 360 [48]. The generated files are usually saved in STL format. This step is foremost important since creating models for simulation constructs the simulation world for training.

To finally launch the model, the exported STL or OBJ file has to be written as a model description in the SDF file which is readable for Gazebo. As mentioned above, the scaling of the imported model is critical as well. Since the models generated in simulation are to duplicate their real world counterparts, the dimension of the generated models has to be in the correct proportion to the real model.

One essential step of creating a model is defining its collision properties with a collision model. Collision models are defined with the physical property of the model that includes mass, inertia and acceleration. Most of the parameters are calculated with a high estimation from the actual robot, but these numbers can be measured with higher accuracy with more engineering labor. Lastly, all the launch files can be collectively launched with a .WORLD file.

For testing purposes, two maps were created: a relatively simple map in a building hallway, and a less structured, more complex maze-like map.

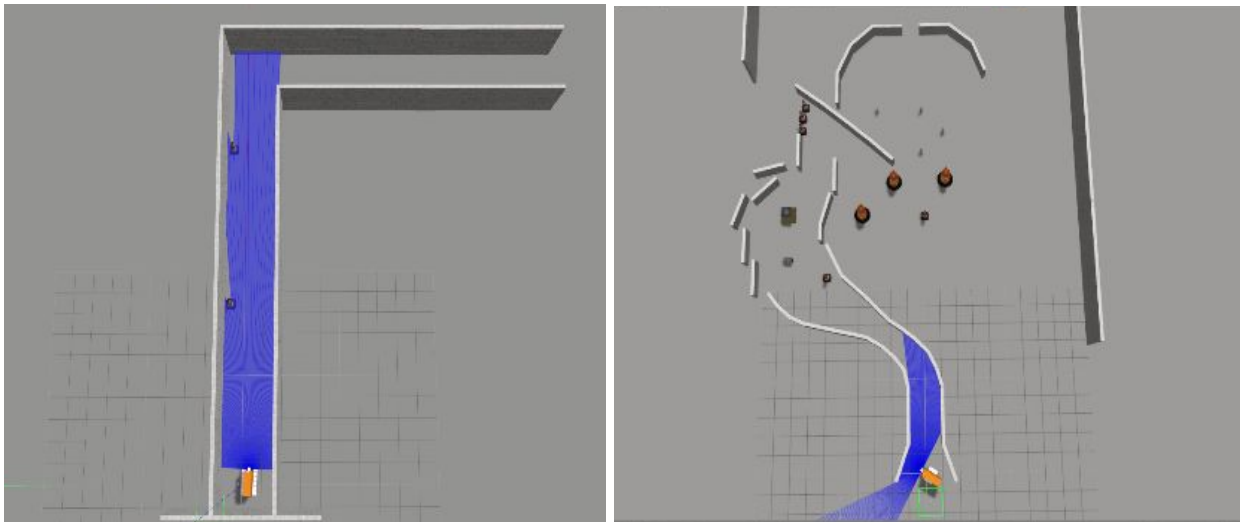


Figure 1.4 Simple testing environment (left), complex testing environment (right)

Figure 1.4 shows the simulated environment with a simple map on the left and complex map on the right. The simple map serves the purpose of testing the progression of the algorithm. In other words, issues can be more obvious with testing in a simple setup. Solving these issues can help to build up a working result that can be further tested with a more complex environment. The variation of obstacle placement, wall shapes and different width of pathways can demonstrate the effectiveness of the algorithms.

The end-goal is to compare the performance of the bug algorithm approach vs a network training approach, as evaluated by running both algorithms in the test environments. .

2 Conventional Bug algorithm

2.1 Wall Following Algorithm

Wall Following navigation refers to driving parallel to a wall or an obstacle while keeping a safe distance from the wall (or other obstacle). This algorithm is extensively implemented for mobile robot navigation [46]. Wall Following can effectively perform obstacle avoidance with an effective controller. When the environment is unknown (but can be sensed, at least myopically), the bug algorithm is the simplest way to navigate while avoiding collisions. While following walls, the robot may try to reach a goal, or it can perform environment mapping. Commonly, a laser scanner is used for detecting distances between the mobile robot and obstacles. The scanned pings can be interpreted as distance from a wall (or other obstacle). The wall-following algorithm should try to maintain a near, but safe distance from the wall (e.g., arbitrarily chosen to be the distance from the robot's left side, resulting in counter-clockwise circulation around obstacles). Controller should act on the perceived distance from the wall to behave appropriately to make progress while maintaining the desired wall-following distance.

The controller must perceive and act on a defined safety range. To visualize the safety region, imagine a thick cover attached to the robot. A threshold to the distance is added so that the robot stays within some tolerance of the safety distance from the wall. For example, if the safety distance was defined as 0.3 meters and the tolerance was 0.05 meters, the robot's distance from the wall can be characterized as: 0.3-0.05 < distance < 0.3+0.5 is "safe", distance< 0.3-0.05

is “(too) close” and distance> 0.3+0.05 is “(too far) away.”

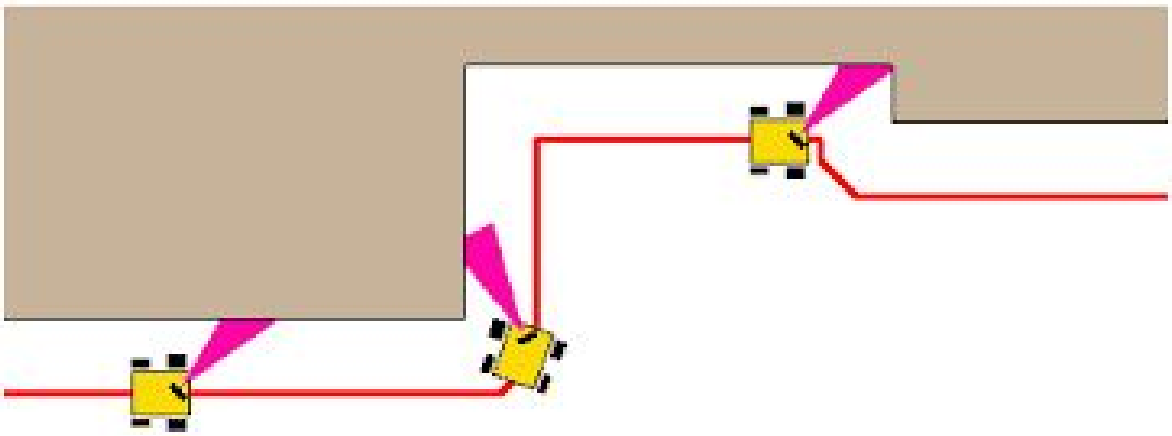


Figure 2.1 Wall following example [40]

Figure 2.1 shows an example execution of the wall following algorithm. The red line draws a pathway the robot should follow to navigate along the walls, and the sensor is constantly checking the distance between the robot and the wall on the left hand side.

2.2 Naive wall-following implementation

The first algorithm was having the robot moving forward and rotating when it sees an obstacle in the front. The laser sensor placed at the front bumper publishes the distance from the scanned pings. Subscribing to the ping that directly looks ahead gives us the perpendicular distance from the front of the robot to the obstacle in the front. The robot would rotate when the front distance is less than 0.3 meters and stop rotating when the front distance is larger than 0.3 meters. The first execution of results can be viewed in the YouTube link [47].

However, this algorithm has issues where the robot hits the wall on the left or right side bumper since seeing only the distance in the front is not sufficient to navigate through the walls.

2.2.1 improved naive wall-following implementation

A step forward is to implement an algorithm that lets the robot follow a wall.

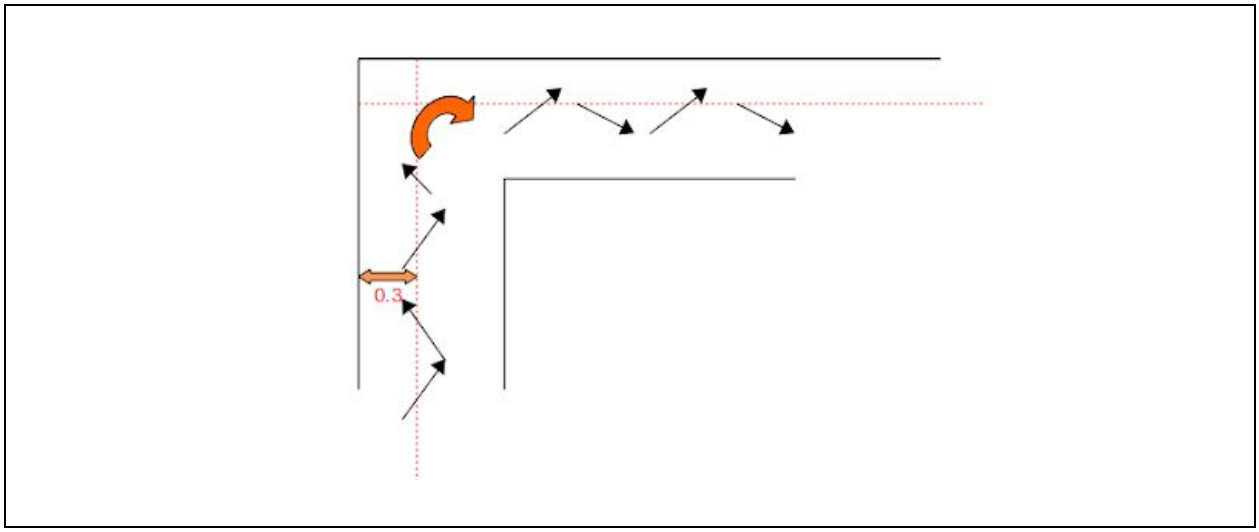


Figure 2.2 Zigzag motion of the robot

Setting a "safe distance" lets the robot keep at a constant distance between its left side to the left side wall. This improved algorithm has the robot move forward until the front sensor sees an obstacle within 0.3 meter. Then the robot rotates until the front sensor clears. An added sensor is overlooking the left front bumper at a 45 degrees angle. This sensor is checking the distance from the left front bumper to the wall and keeps a 0.3 meters from the wall. The distance from the left front bumper to the wall is called df. If df is less than 0.3 meters, the robot rotates clockwise 20 degrees and moves forward. If df is more than 0.3 meters, the robot rotates counter-clockwise 20 degrees and moves forward. The forward velocity is 0.1 m/s. This results in a zigzag motion shown in figure 2.2. To improve the continuity of the navigation, a wall following controller is implemented.

2.3 Improved Wall Following Controller

A controller is integrated with the driver node that estimates the turning angle to keep the robot at a constant parallel offset from the left wall given the front wall state is always available. L is the distance between two wheels. □ is the angular position of the robot with respect to x and using □, the offset of x and y can be calculated from equation (1) and (2).

$$x_{offset} = \frac{\gamma + \gamma_{des}}{2} cos(\square) \quad (1)$$

$$y_{offset} = \frac{\gamma + \gamma_{des}}{2} sin(\square) \quad (2)$$

$$\gamma_{err} = \gamma - \gamma_{des} \quad (3)$$

$$\psi_{err} = \frac{d_r - d_f}{L} \quad (4)$$

Equation 1 and 2 shows the wall offset and offset angle from the robot to the wall.

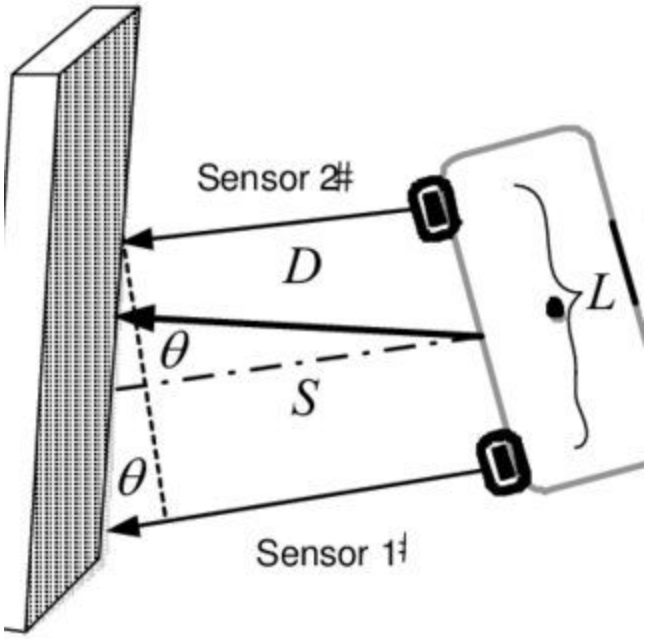


Figure 2.3 Offset angle and offset distance [33]

$$v_{cmd} = constant \qquad (3)$$

$$\omega_{cmd} = K\,offset \cdot offset\ error + K_{\psi} \cdot angle\ error \qquad (5)$$

$$\omega_{cmd} = K_{\gamma} \cdot \gamma_{err} + K_{\psi} \cdot \psi_{err} \qquad (6)$$

$$\omega_{cmd} = (-K_{\gamma} \cdot (\gamma - \gamma_{des}) - K_{\psi} \cdot sin(\frac{d_r - d_f}{L}) + K)\, v \qquad (7)$$

The variables d_r and d_f are the rear left-side distance and front left-side distance from the robot to the left wall, respectively. ψ_{err} is the offset angle that needs to be adjusted to zero as well as the offset distance. In equation 4, K is a control parameter that is intended to incorporate using knowledge of the desired path curvature. However, this parameter can also be used to bias the robot to try to curve to the left, which can be helpful when trying to do wall following with the wall on the left. [33]. The result can be viewed in the YouTube Link[43].

2.4 Adjustment from the controller

The controller showed a steady and constant wall following execution. However, it lacked the intelligence to navigate through narrow passages. The first testing did not include any traffic cones in the map. But, once the cone was introduced to the environment, the robot behaved as illustrated in figure 2.4.

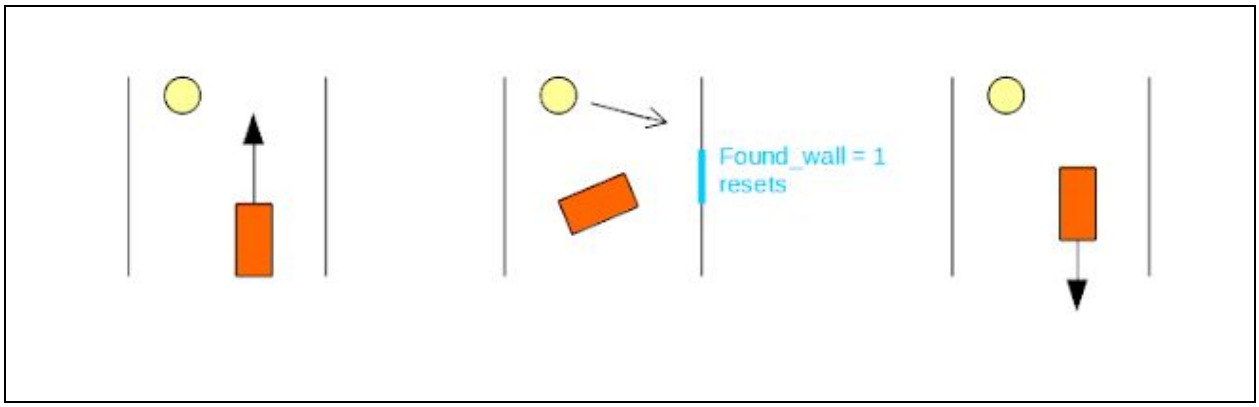


Figure 2.4 Unsuccessful passing the obstacle

At instances, the robot sees the cone as a new wall then spins around until it directly faces the "East" wall. Then the robot resets its 'Found_wall' boolean as 1 again and considers this wall as the wall to be followed. The robot then continues to rotate to orient itself such that the new wall is on the robot's left hand side. This results in the robot heading in the opposite direction, then ultimately spinning around the lower boundary of the wall, then following this wall on its opposite side. With testing and tuning of the parameters, when curvature K is set to be at 3 and forward speed set to 0.2 meters per second, the robot can successfully bypass the cone.

2.5 Wall Following Controller Integrated Results

The Wall Following Controller results can be viewed on YouTube Link [43]. Figure 2.5 shows the robot successfully navigating past the cone placed near the left wall.

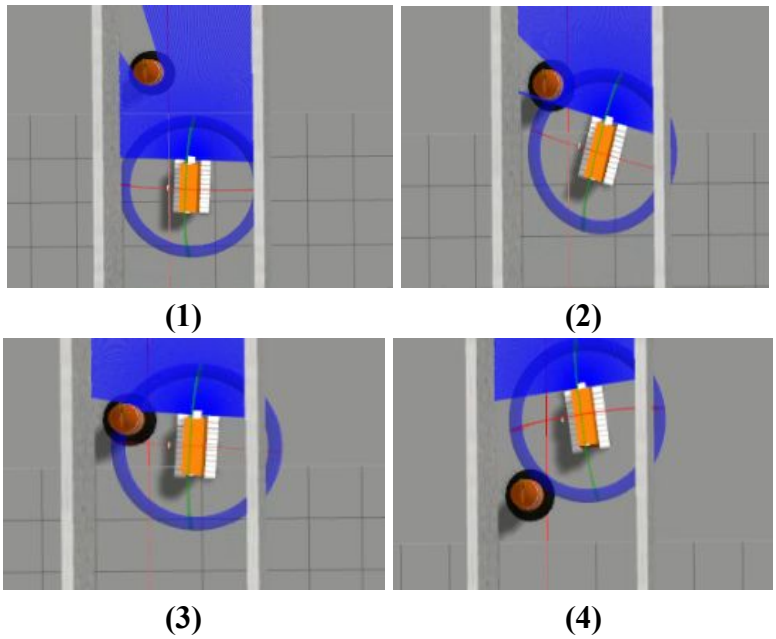


Figure 2.5 Wall following controller results in still images

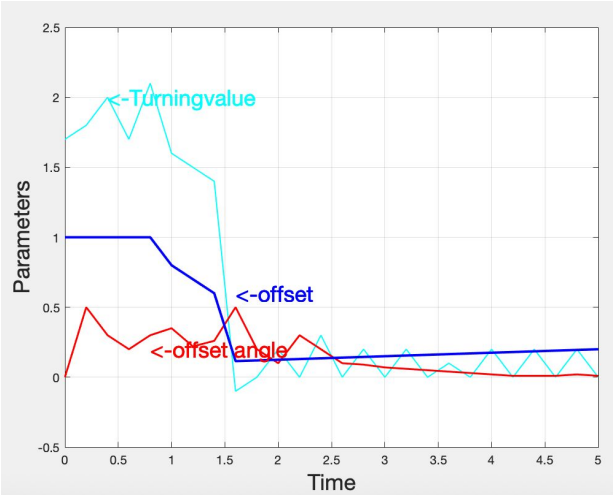


Figure 2.6 Parameter output of cone bypass [33]

Figure 2.6 shows the parameter adjusted corresponding to the bypass of a cone on the left hand side of the wall. The term turning value is the omega command, offset is γ_{err} from equation (1) and offset angle is ψ_{err} from equation (2). From the graph, the offset drops from 1 to 0.15

and keeps a continuous wall following. The omega shows the robot is gradually turning left and with a constant offset at 0.3 meters, the robot is following the wall on the left hand side.

3 Neural Network Approach

3.1 Convolutional neural network

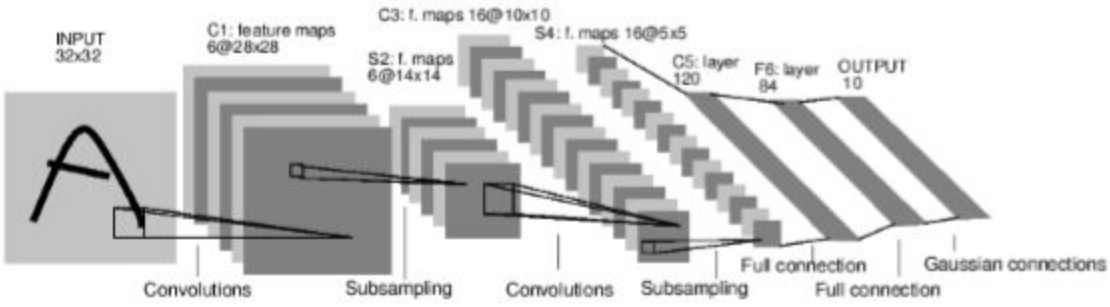


Figure 3.1 The structure of a convolution neural network LeNet-5 [24]

As early as the year of 1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts created a neuron model that became the foundation for future neural networks [34]. In 1958, Frank Rosenblatt designed his first neural network called the Perceptron. This network was to perform simple classifications with decision boundaries. Over the years, new realizations, models and variations were introduced to artificial neural networks, and interest grew, waned, and ultimately rebounded. Artificial neural networks, particularly "deep" networks, are now mainstream in machine learning and of great interest in robotics [34].

A “deep” network has many layers of neurons. A “convolutional” network uses a pattern of weights that is replicated within the network, thus allowing a large number of synaptic connections with a relatively small number of free parameters. The repeated patterns of weights are “kernels”, and these kernels can be “trained” (assigned specific parameter values) such that the kernels instantiate feature detectors [24]. Multiple kernels are defined as operations between successive layers of neurons. Discovery of what constitutes good kernel values is the essence of “learning” or “training” in a convolutional neural network. Each layer of the network receives stimuli from kernels acting on lower, more primitive layers, yielding higher levels of abstraction in ascending layers. Additional operations that have been found useful include nonlinear activation functions and “pooling”. In theory, more layers constructed could result in a more intelligent network. However, this is at the cost of requiring more training data, potentially much longer training times, and slower execution of deployed networks. For this research, there are six layers and the architecture of the network used for this research is LeNet-5 developed by Yann Lecun, Yoshua Bengio, Leon Bottou, and Patrick Haffner [24].

3.2 Perceptrons

Perceptron is an algorithm that constructs learning for neurons just as their biological counterparts. This is a breakdown of the neural network in terms of a single spiked neuron with respect to the inputs x1,x2, xn and weights ω_i where ‘i’ indicates which layer the weights are in.

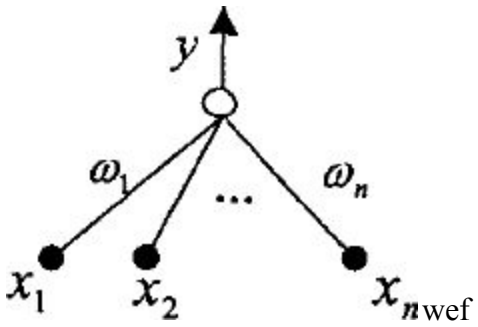


Figure 3.2 Single-layer perceptron [18]

In terms of formula expression, the graph can be written as [18],

$$y = f(\sum_{ii} - \Theta)$$

$f(\cdot)$ is the activation function and $_i$ is the i'th input to the perceptron, which is weighted with gain $_i$ [18]. The inputs are independent variables, so the output is mostly decided by the weights that are trained from the network. "Training" is conducted by altering the weights to attempt to have the network become a function that correctly maps input patterns (values of $_i$) onto target values of y. This requires a potentially large set of examples of input patterns with corresponding output values. This also requires a strategy to alter the weights to achieve improved success rates in having the perceptron map inputs into outputs. A simple method for updating weights is as follows [24]::

$$w_i' = w_i - \eta . \frac{dE}{dw_i} \qquad (50)$$

ω'_i is the weight to be updated, η is the learning rate, $dE/d\omega_i$ is the target value, y is the output value from the most recent iteration with candidate weights, and $_i$ as the i^{th} element of the input pattern. This updating process ceases when the linear regression comes to a sufficiently small error (or ceases to improve) [20]. This approach, using a single perceptron, can be

successful in categorizing input patterns, provided the input patterns are linearly separable.

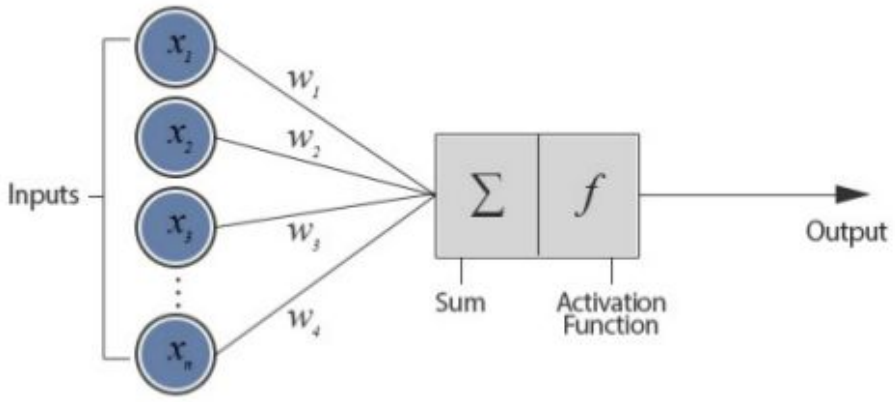


Figure 3.3 Structure of perceptron [19]

3.3 Back propagation

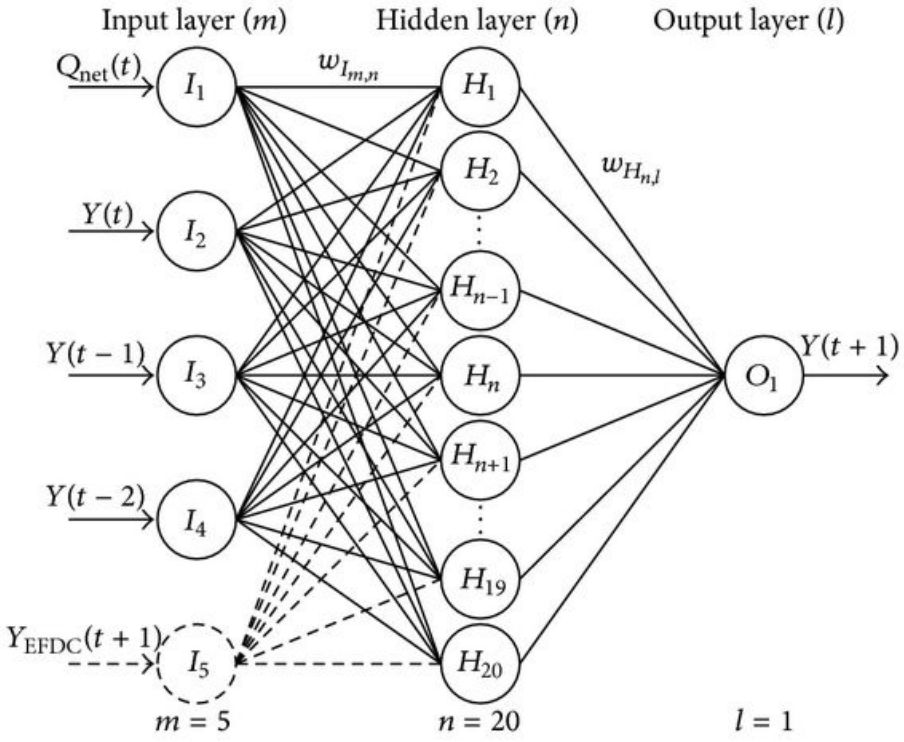


Figure 3.4 Fully connected feedforward neural network [15]

Figure 3.4 shows the layout of a feedforward artificial neural network. In this network, the outputs of each layer become the inputs of the next layer. While all the neurons are interconnected, this is done in a structured, feedforward organization. Any of the layers between the first input layer and the last output layer is called a hidden layer. The goal of the network is to yield a useful output for any given input pattern, and the hope is that the network succeeds in generalizing from examples. The approach to attempting this is to minimize the errors associated with a "training set" of examples for which the correct mapping is known, where minimization is achieved by varying the synaptic weights. This constitutes an optimization problem, where the weights are the parameters to be optimized and the network output errors for the training set are to be minimized. This is commonly performed through a gradient descent algorithm known as back propagation [35]. The error metric used for optimization is sometimes referred to as the "loss".

The training algorithm updates the weights between connected neurons in order to reach the minimum loss. For this research, the input patterns considered are images, deliberately cropped and downsampled to a 32 by 32 pixel pixel array. Typically, pixel values represent intensity of gray-scale images, or RGB values of color channels. With respect to collision sensing, however, color is irrelevant. Instead, to represent the depth information, the pixel values are used to encode depth. In this manner, networks designed for 2-D images can be applied to 3-D data.

While Fig 3.4 represents a generic feedforward network, Fig 3.2 illustrates a specific pattern to the weights. When a 2-D image is vectorized, the spatial relationships in the 2-D

structure may get lost. However, by using convolution kernels, these spatial relationships are preserved, and the output of each convolution can be treated as the input to another 2-D image.

This process is repeated for each layer, with a different set of convolution kernels between successive layers. For each candidate set of weights, each of the training-data input patterns is imposed on the network, and the corresponding computed output values are compared to the "correct" answer. To train the network, the weights are adjusted, seeking to reduce the overall network error over the complete training set. Backpropagation is commonly used to perform such training [16]. If it is known that the input patterns are related temporally, it can be useful for the network to include some short-term memory, e.g. as below [17]:

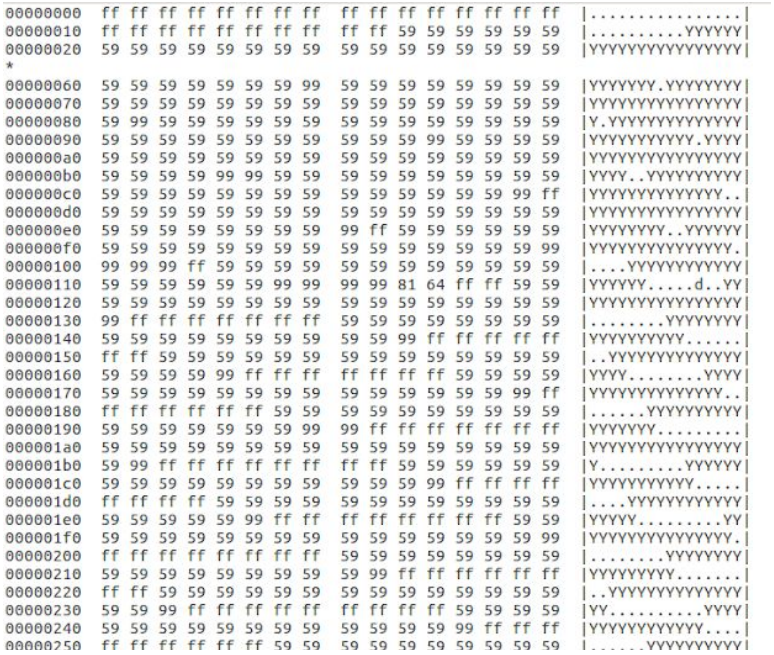
3.5 Network & ROS communication

Network and ROS communication is key for deploying neural networks with ROS-controlled robots. The method of this communication is through TCPROS [36]. TCPROS is a layer for ROS 'messages' and 'service' communication. ROS 'messages' are published without regard to which other nodes might "subscribe" to these messages [37]. In contrast, client/server communications are peer-to-peer with guarantees on receipt of client requests. This infrastructure helps with the design of complex systems by establishing simple interfaces.

Neural Networks were not originally envisioned as part of ROS. To integrate a network with ROS, it can be structured as follows. The network can be encapsulated within a ROS wrapper and instantiated as a service. A client of the service can send an input pattern to the service (network), and the service can perform network computations and return the computed result. Alternatively, the network could be instantiated as a "listener", which receives input

patterns and publishes computed outputs. In the present research, a node called ‘Driver.cpp’ subscribes to the topic published from the network’s output, and the driver node uses this output to control navigation. This is the general communication based on this service and client pair nodes.

Multi-path TCP deployment is the method that sets up the connection between networks and the navigation node [38]. The networks are connected to the node through TCP socket proxies while subscribing to the sensor output simultaneously. This socket path bridges between TensorFlow are written in Python and ROS nodes are written in C++. Currently there are more methods for network deployment with more updated TensorFlow available. A TCP connection requires a unique ‘HOST’ and ‘PORT’ for client and server to set up a connection. If the network tries to subscribe to a certain topic, the server ID from the publisher needs to set its IP address the same as the HOST. Since there are more than one network, the server ID has to be set up with a floating string based on whichever network is called back at the moment. The data format of the processed point cloud snapshot was chosen as the 2D matrix where the z dimension is representing the pixel values of the snapshot.



```
00000000  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
00000010  ff ff ff ff ff ff ff ff  ff ff 59 59 59 59 59 59  |..........YYYYYY|
00000020  59 59 59 59 59 59 59 59  59 59 59 59 59 59 59 59  |YYYYYYYYYYYYYYYY|
*
00000060  59 59 59 59 59 59 59 99  59 59 59 59 59 59 59 59  |YYYYYYY.YYYYYYYY|
00000070  59 59 59 59 59 59 59 59  59 59 59 59 59 59 59 59  |YYYYYYYYYYYYYYYY|
00000080  59 99 59 59 59 59 59 59  59 59 59 59 59 59 59 59  |Y.YYYYYYYYYYYYYY|
00000090  59 59 59 59 59 59 59 59  59 59 59 99 59 59 59 59  |YYYYYYYYYYY.YYYY|
000000a0  59 59 59 59 59 59 59 59  59 59 59 59 59 59 59 59  |YYYYYYYYYYYYYYYY|
000000b0  59 59 59 59 99 99 59 59  59 59 59 59 59 59 59 59  |YYYY..YYYYYYYYYY|
000000c0  59 59 59 59 59 59 59 59  59 59 59 59 59 59 99 ff  |YYYYYYYYYYYYYY..|
000000d0  59 59 59 59 59 59 59 59  59 59 59 59 59 59 59 59  |YYYYYYYYYYYYYYYY|
000000e0  59 59 59 59 59 59 59 59  99 ff 59 59 59 59 59 59  |YYYYYYYY..YYYYYY|
000000f0  59 59 59 59 59 59 59 59  59 59 59 59 59 59 59 99  |YYYYYYYYYYYYYYY.|
00000100  99 99 99 ff 59 59 59 59  59 59 59 59 59 59 59 59  |....YYYYYYYYYYYY|
00000110  59 59 59 59 59 59 99 99  99 99 81 64 ff ff 59 59  |YYYYYY.....d..YY|
00000120  59 59 59 59 59 59 59 59  59 59 59 59 59 59 59 59  |YYYYYYYYYYYYYYYY|
00000130  99 ff ff ff ff ff ff ff  59 59 59 59 59 59 59 59  |........YYYYYYYY|
00000140  59 59 59 59 59 59 59 59  59 59 99 ff ff ff ff ff  |YYYYYYYYYY......|
00000150  ff ff 59 59 59 59 59 59  59 59 59 59 59 59 59 59  |..YYYYYYYYYYYYYY|
00000160  59 59 59 59 99 ff ff ff  ff ff ff ff 59 59 59 59  |YYYY........YYYY|
00000170  59 59 59 59 59 59 59 59  59 59 59 59 59 59 99 ff  |YYYYYYYYYYYYYY..|
00000180  ff ff ff ff ff ff 59 59  59 59 59 59 59 59 59 59  |......YYYYYYYYYY|
00000190  59 59 59 59 59 59 59 99  99 ff ff ff ff ff ff ff  |YYYYYYY.........|
000001a0  59 59 59 59 59 59 59 59  59 59 59 59 59 59 59 59  |YYYYYYYYYYYYYYYY|
000001b0  59 99 ff ff ff ff ff ff  ff ff 59 59 59 59 59 59  |Y.........YYYYYY|
000001c0  59 59 59 59 59 59 59 59  59 59 59 99 ff ff ff ff  |YYYYYYYYYY.....|
000001d0  ff ff ff ff 59 59 59 59  59 59 59 59 59 59 59 59  |....YYYYYYYYYYYY|
000001e0  59 59 59 59 59 99 ff ff  ff ff ff ff ff ff 59 59  |YYYYY.........YY|
000001f0  59 59 59 59 59 59 59 59  59 59 59 59 59 59 59 99  |YYYYYYYYYYYYYYY.|
00000200  ff ff ff ff ff ff ff ff  59 59 59 59 59 59 59 59  |........YYYYYYYY|
00000210  59 59 59 59 59 59 59 59  59 99 ff ff ff ff ff ff  |YYYYYYYYY.......|
00000220  ff ff 59 59 59 59 59 59  59 59 59 59 59 59 59 59  |..YYYYYYYYYYYYYY|
00000230  59 59 99 ff ff ff ff ff  ff ff ff ff 59 59 59 59  |YY.........YYYY|
00000240  59 59 59 59 59 59 59 59  59 59 59 59 59 99 ff ff  |YYYYYYYYYYYY....|
00000250  ff ff ff ff ff ff 59 59  59 59 59 59 59 59 59 59  |......YYYYYYYYYY|
```

Figure 3.5 .data file in collected data folder

Since the goal of pursuing a neural network application is to achieve robotic navigation with the feedback from the networks alone, the conventional and the network-based algorithm share the same driving controller. Essentially, the conventional algorithm listens to pings of the robot's position from the laser scanner, and uses these parameters to adjust the robot's behavior for wall following navigation. Similarly, the networks compute these parameters with the input images from the sensors based on the pre-trained weights. The PC on which this research was carried out had 16Gs of RAM, AMD Ryzen eight-core processor with a GeForce RTX 2070 graphics card. With such computational power, the computational time between the network powered by TensorFlow and the ROS bridge takes about 1.5 seconds. The sensor built on the robot is a Kinect sensor that publishes 640 x 480 = 1,228,800 points at each batch [24]. In order to pre-process the points, a voxel grid filter was used to divide the 3D published cloud into cubes to record a more uniform format of data. When the resolution of the recorded images gets cut

down, the computational time and training period go down as well. The prediction rate could be high as 1,800 images per second with a computational time of 0.75 seconds [24].

The network is constructed and written in Python with TensorFlow, and the driving controller is written in C++ with ROS. As mentioned before, the network and ROS communicate over TCPROS client-server architecture.

3.6 Training setup and data collection

Constructing a well trained neural network requires proper data collection and labelling. The amount of data needed to be collected is too large for manual collection. The data generation is fully automated for this research through the use of simulation.. Acquiring a large set of data is time consuming. Further, the training data must be sufficiently rich, experiencing a variety of environments and experiencing both safe and unsafe scenarios. A Gazebo simulation was created in which the robot could sense the environment and experience collisions (or no collisions), providing a means to label the sensory data.

One challenge is the imbalanced number of training data from each labeled folder. The positive set will contain much more batches of images compared with the images in the negative labelled set. This is because only a few images are recorded as negative right before the collision, and the rest of the images would be positive. On average, this would result in a high accuracy for classifying positive inputs and causing the classifier to ignore the negative inputs with a low network sensitivity. Thereafter, the training would not be effective if the network only recognizes the positive labelled images. The drop-out ratio was later calculated in this thesis

showing how the data was artificially manipulated to reach a balanced ratio between the positive and negative labelled sets.

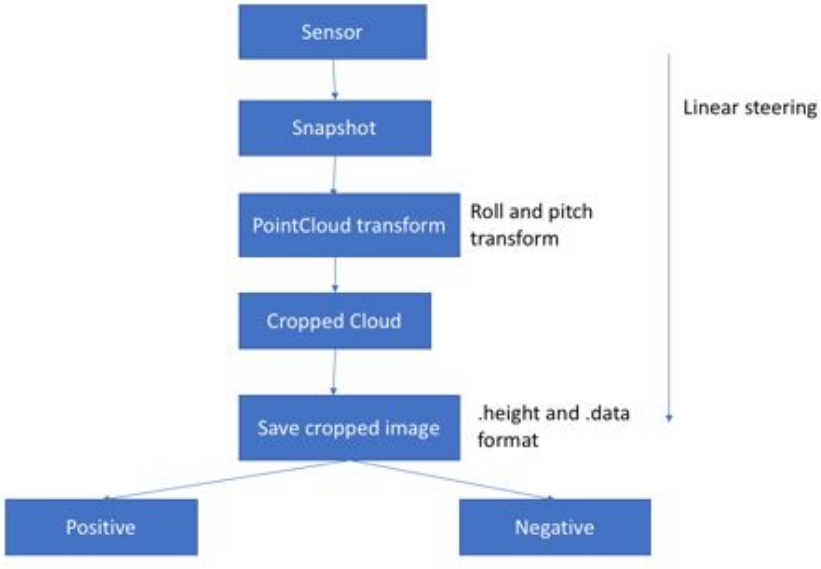


Figure 3.6 Outline of data generation and labelling

Figure 3.6 shows the outline of the data collection process. The linear steering arrow on the right of the diagram represents the collection process taking place while the robot is moving forward, rotating or stationary. The sensors used on the robot were simulated Kinects. A raw 3D point cloud was published by each Kinect sensor on a respective ROS topic, and the data included listing x,y and z coordinates of each point pinging in the environment. In order to utilize an existing 2D image-based network architecture, the 3-D point cloud was pre-processed to represent it such that the gray-scale intensity was proportional to depth. In this way, depth information was preserved, at the expense of losing color information. However, color should not be relevant in the context of evaluating potential collisions.

After a week-long automated data collection process, roughly 67,887 images of positive labelled and 68,738 negative labelled images were collected. This is after manually omitting the majority of repeated and positive labelled images. The positive labelled images (i.e. those for which an impending collision was to be predicted) were collected at 0.15 meters prior to collision.

```
while true {
    Spawn the model in a random location in the map facing a random direction;
    Generate a straight line extending along the forward direction of travel;
    while a driving failure is not detected {
        Send a steering command to move forward, keeping the model center as \
            close as possible to the line;
        if the model has progressed 30cm or more since the last cropping {
            Crop a 4-meter corridor section from the current point cloud;
            Process the cropped point cloud section into a heightmap image and \
                save to the list along with the model's current location;
            for each heightmap image in the list {
                if the current model location is within the heightmap image {
                    crop a small section corresponding to the model location out \
                        of the heightmap image and save it with a positive label;
                }
            }
        }
    }
    for each heightmap image in the list {
        if the current model location is within the heightmap image {
            crop a small section corresponding to the model location out \
                of the heightmap image and save it with a negative label;
        }
    }
    clear the list of saved heightmap images;
}
```

Figure 3.7 pseudocode of data generation [24]

More specifically the collector code ran as follows. Once the robot was spawned at a random location with random orientation, a linear steering controller was called that commanded the robot to move forward. Data was collected with each processed snapshot saved in short-term memory. The robot kept driving forward until the robot experienced a failure mode. A failure mode was defined as falling off the map, colliding with a wall or flipping over. The machine state was monitored to detect a failure mode and once triggered, all sensor data in short-term

memory was interpreted as predictors of “safe” or “unsafe” conditions. The linear command velocity was set at 0.5 meters per second. Images were acquired every 30 cm of distance travelled. These snapshots are cropped, reduced to a lower resolution, and saved in a short-term memory for temporary storage. Once a collision or failure was detected, the saved images were interpreted. Images associated with a “far” distance from collision were labelled “safe” (or “negative”), and images associated with a “near” distance to failure were labelled as “unsafe” (or “positive”), where “near” and “far” were defined by an arbitrary safety-distance threshold. This process was repeated automatically. After each failure, and subsequent classification of recalled images, the robot would respawn at another location on the map with a random position and orientation. This process was repeated continuously for a week-long collection period.

The data collection process can be viewed from the YouTube Link [42]. This video shows the robot spawned at a random position on the map with random orientation and running forward until a bumper detected a collision.

4 Training Results

4.1 Sensor Adjustment

The initial design of the robot had only one Kinect sensor on the front top of the robot. This sensor alone was enough to achieve many functionalities including navigation and obstacle detection. However, the conditions of an indoor environment can be more challenging than that of an outdoor one, particularly when making progress requires navigating through a narrow

passage. Imagine the left map from figure 1.2 If the robot were asked to navigate through such open environments, a front bumper sensor would be sufficient. The terrain is open enough that, when the robot detects forward motion is dangerous, it could respond by rotating to another heading. However, when considering narrow passages, the robot must tolerate skirting danger and it must avoid excessive rotations. This motivates using wall-following as a navigation strategy. However, wall following also requires consideration of both distance from a wall and relative angle to the wall. Thus, side sensors are required in addition to a forward sensor.

4.2 Network structure

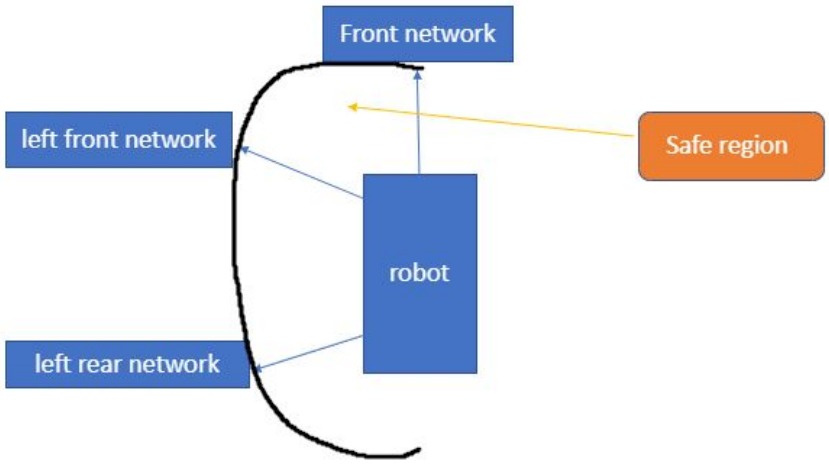


Figure 4.1 region for wall following with network controller

Figure 4.2 shows the region of the 'safe distance' predicted from the network sensor. The network was trained based on labelled data of 'safe' and 'unsafe', and in this case the 'safe' is all

the images taken further than 0.15 meters prior to collision and 'unsafe' images are those acquired within 0.15 meters of collision.

While there are many variations of strategy one could use for network-based wall following, in the present work, a particularly simple approach was considered. The trained network for the forward sensor was capable of predicting impending collision in the forward direction, with binary outputs of "safe" or "unsafe".

The side sensors were intended to achieve a desired offset from a wall and heading parallel to a wall. These were implemented, though, as exact replicas of the forward sensor. A rationalization for this is as follows. From the viewpoint of a side sensor, the network could predict that, if the robot reoriented to the viewing angle of that sensor, it would be dangerously close to hitting a wall or not close to hitting a wall. In essence, then, the side sensor's network indicates if the robot's side is close or not close to a wall. With two such side sensors--front and back--the close/not-close (i.e. safe/unsafe) network interpretations can be combined to estimate if the robot is both desirably close to a wall and roughly parallel to the wall. Thus, it seems plausible that the single trained network could be used for interpretation of all three sensor streams.

```
if the front sensor = unsafe{
    wall found = true;
    spin clockwise until front sensor = safe
    follow the wall on the left
}
while (front sensor = safe){
    if left front or left rear sensor = unsafe{
        wall found = true;
        follow the wall on the left
    }
    if both front =safe & rear left = safe{
        move forward and the robot is seeking the wall
        spin counterclockwise until front left = unsafe //given conditions where the wall encounters a sharp left turn
    }
    if front left = unsafe & rear left = safe{
        move forward & spin clockwise as a ramp motion
    }
    if front left = safe & rear left = unsafe{
        move forward & spin counterclockwise as a ramp motion
    }
    if front left = unsafe & rear left = unsafe{
        move forward & spin clockwise as a ramp motion until front left = safe
    }
}
```

Figure 4.2 Pseudocode of wall following controller with analogue network callback

Figure 4.2 shows the pseudocode of the network controller that uses the three network outputs to achieve wall following. The hierarchy is as follows. First, the front network sensor is checked. If The front sensor indicates "unsafe", then the robot spins clockwise until the front sensor indicates "safe." If the front sensor indicates "safe" to move forward, then the side sensors are consulted. These sensors are used to modify the robot's heading to try to maintain a desirable distance from an obstacle to the left, as well as achieve a robot heading that is tangent to the obstacle on the left. This is determined by a combination of "safe" and "unsafe" evaluations by the two side sensors. Behaviors under the various cases are listed in Fig 4.2.1.

Note that to move tangent to an obstacle, both side sensors should be on the borderline between "safe" and "unsafe". The safe distance trained from the data collection is 0.15 meters prior to collision, and with this information, the robot toggles between safe and unsafe to keep a nearly constant distance from the wall. Instead of calculating offset angle and distance, wall following is directed by the 'safe' and 'unsafe' outputs. In the present implementation, the robot

can “chatter” between safe and unsafe evaluations, which can lead to unsmooth motion. However, that is tolerated in the present work for the simplicity of using a single trained network with a binary output. Smoother implementations are left to future work.

4.3 Network controller results

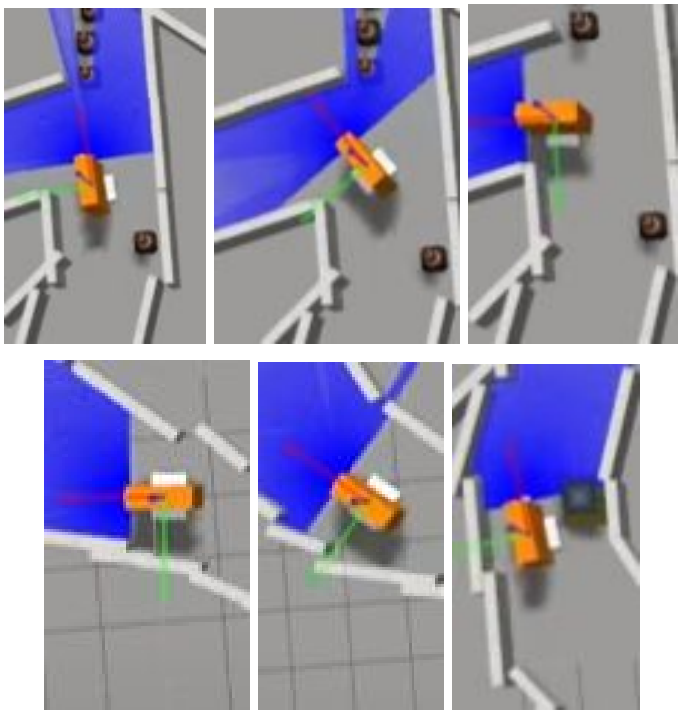


Figure 4.3 Network controller results

A precursor approach with the network controller used side sensors on both sides of the robot. The proposition was to get the outputs from all sensors and whichever one reported negatively, the robot would spin the opposite way. This way, with a full safety contour surrounding the robot, the robot could always be kept at the center between walls. However, this approach had two issues. The first issue was that this implementation did not conform to a bug algorithm, since it would follow a wall on either side. As a consequence, the behavior lost the assurances of maze-solving capabilities of a simple bug algorithm. Nonetheless, the robot

exhibited useful behavior. If subgoals could be placed intelligently, the robot was capable of navigating through narrow passages. A second issue with this implementation was that the robot would ignore turns and corners. Figure 4.3 shows a sharp left turning corner. If the controller followed both walls, then it would go all the way up to the end of the tunnel and spin back around then drive back to the starting position.

To solve these issues, the network controller was designed to follow contours only on the robot's left. With this version, the robot successfully passed the sharp left corner as shown in figure 4.3, as can be seen via YouTube Link here [49]. The video link shows the robot was able to pass through a narrow, irregular hallway with broken walls, avoiding obstacles like cones on the way and turning at the left corner.

4.4 Training results interpretation

Given the equation of calculating the training accuracy as

Accuracy = (number of positive labelled + number of negative labelled) / (number of positive labelled + number of negative labelled + number of dropped out positive + number of dropped out negative) *[24]*

After preprocessing each depth image, the output was condensed to a 32x32 gray-scale image, with intensity representing distance. In the convolutional network, each kernel had a size of 2 by 2. It should be noted that the ratio of "safe" to "unsafe" labelled data acquired during the data collection phase was 90/10. With such unbalanced training sets, the network would be biased to label images as "safe", and thus it would err on the side of failing to recognize impending collisions. To counteract this, the "safe" set was reduced to balance it with a comparable number of "unsafe" examples. The training was performed with a set of 6,000,000

images that contained a balanced set of “safe” and “unsafe” labels. Training was done incrementally (i.e. batch size of 1), which took almost 17 hours to complete. Loading the input images and training from the loaded CSV file was the chief bottleneck.

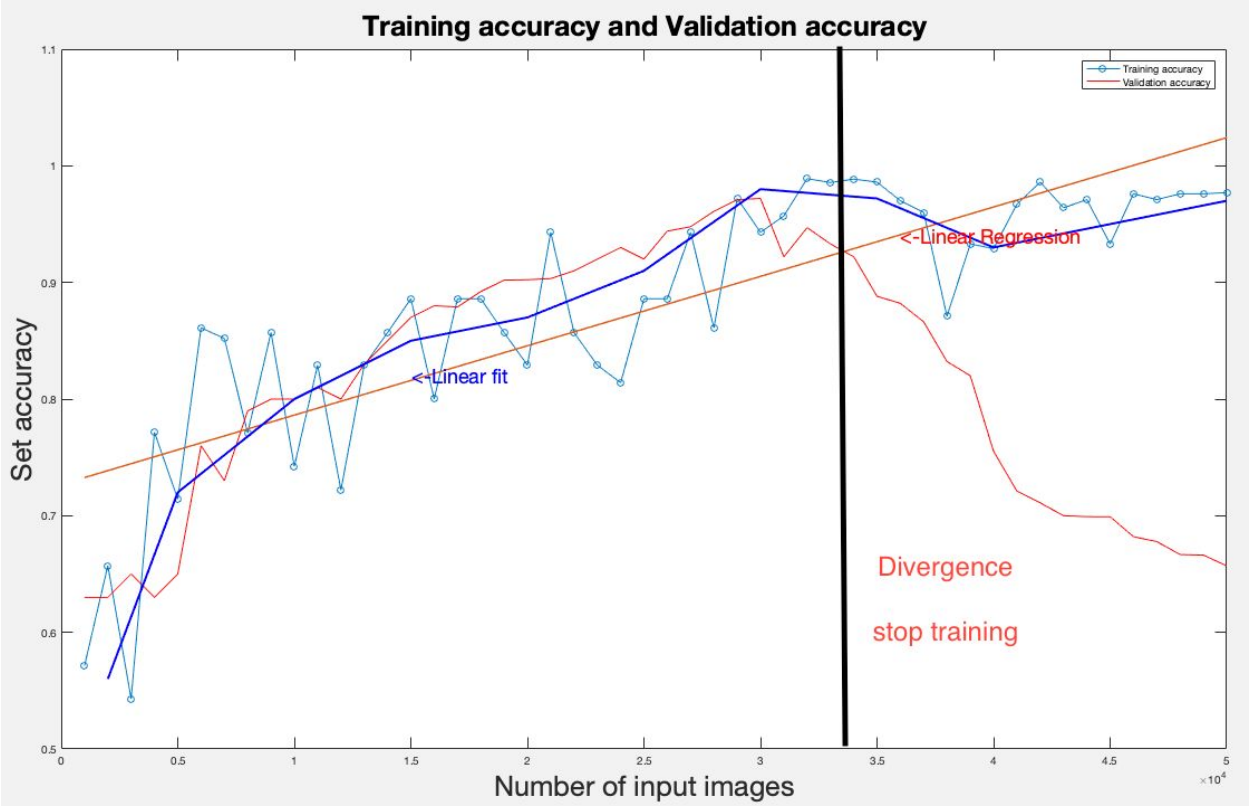


Figure 4.4 training accuracy and validation accuracy set

Figure 4.4 shows the progression of training accuracy and validation accuracy. Pinning down an exact point at which to cease training is not obvious. Overtraining could occur if the training does not stop appropriately.. This problem is avoided by use of a validation set. The validation data set is taken separately from the training data set. If the validation accuracy starts to decrease then it indicates the network is encountering overtraining, and the training should stop at where the training accuracy and validation accuracy diverge. In figure 4.4 the vertical

black line shows the point of divergence at 3,400,000 images, at which point the network had achieved a 97.85 % accuracy.

5 Results and analysis

Using network-based interpretation of depth-camera data in the algorithm of Fig xxx, the robot was tested navigating in complex environments. As will be discussed in Chapter 6, completion of the Bug algorithm for maze solving requires addition of an "M-line" (a default initial straight-line path from start to goal), and this line can be treated as a virtual obstacle. However, bug algorithms assume no prior information of the environment. Often, much can be known about the environment, e.g. from building floor plans. At the same time, a robot cannot operate safely on the basis of a priori information. Obstacles will be present that are not part of a floorplan, including pedestrians, furniture, open or closed doors, etc. It is thus necessary that the robot can react appropriately to its sensors while navigating. Wall following can be combined with coarse planning to achieve effective navigation. For example, based on a floor plan, one could specify a sequence of nominal subgoals. The robot could attempt to achieve these subgoals while navigating around perceived obstacles.

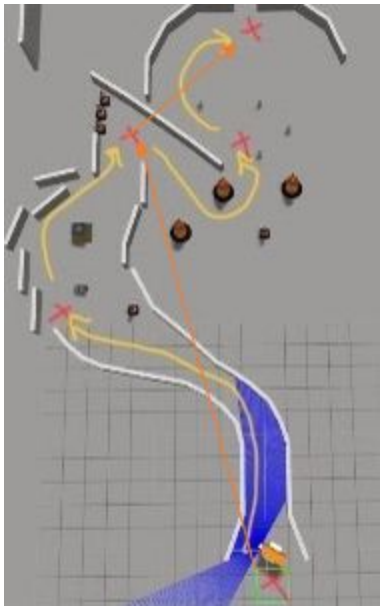


Figure 5.1 wall following with subgoals

The sub-goals implementation can be seen in figure 5.1 where the sub-goals are indicated by the red crosses. In a complex setup such as the one in figure 5.1, the robot would have a tough time navigating to a single end goal. Therefore, having sub-goals further helps the robot to navigate by a pre-planned pathway. The neural network controller helps to avoid obstacles, walls and even moving subjects. So, the use of a neural network controller is still very meaningful in terms of indoor navigation.

Both the conventional wall-following algorithm implementation and the CNN-based wall-following implementation behaved well in simple environments. Compared to the conventional implementation, the network implementation was computationally demanding, including CPU, GPU and network bandwidth.

However, the network-based controller showed some promise of improvement over the explicit algorithmic wall-following implementation in complex environments. Some examples are shown via YouTube here [49].

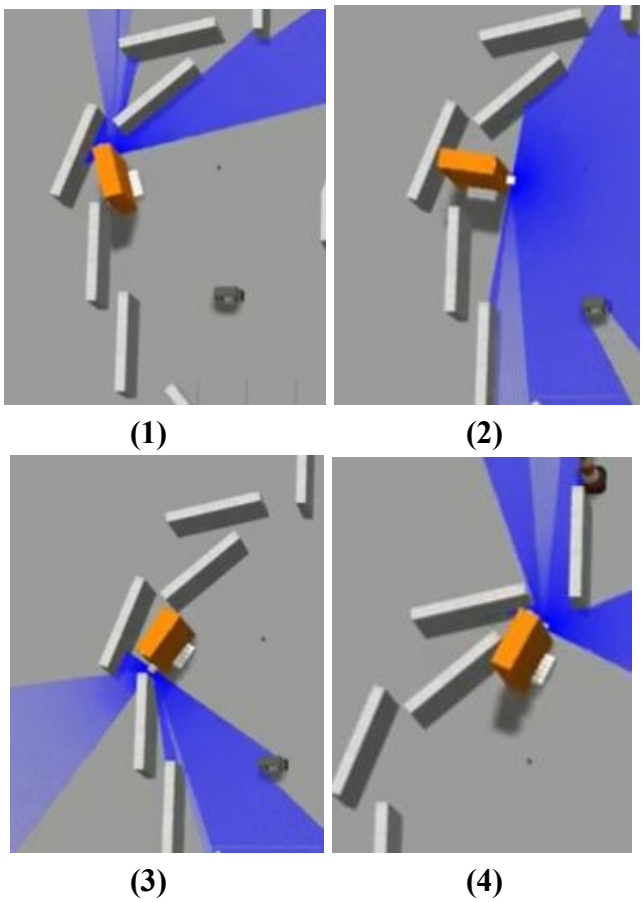


Figure 5.2 Stills of the Conventional results in the complex setup

An example in which the conventional algorithm struggled is shown in Figure 5.2. In this case, the walls had narrow gaps where the robot could get stuck. The conventional wall-following algorithm struggled with getting stuck, or giving up on viable narrow passages, or spinning around inappropriately. This issue was not obvious with the network controller. The network output gave a much more generalized prediction of safe or unsafe motions from the input images. While much improvement could still be made, the network-based controller indicated capability of making good choices in novel, complex scenarios.

The present results are suggestive, not definitive. Nonetheless, observations suggest that a neural-net based controller shows potential for making intelligent decisions in navigation based on depth-camera data. The trained network indicates a high accuracy in predicting impending collisions. Such training has the potential to avoid collisions that might not otherwise be recognized by an explicit algorithm. Using training based on full 3D imaging, the robot would not be fooled by desks, saw-horses, clotheslines, and other obstacles that can give the illusion of safe passage if only a fixed-height horizontal scan were used. Also, low obstacles, such as bricks or blocks that should not be driven over, can be interpreted from 3D images but may be missed by simple LIDAR-scan interpretations. Narrow passage that can and should be traversed, such as doorways, might get rejected by a conventional image interpretation but accepted by a network-based interpretation. Consideration of collisions when rotating a non-circular robot are difficult to incorporate explicitly without resorting to a higher-dimensional configuration-space mapping, which increases complexity and potential additional programming errors. In contrast, the network-based approach makes its evaluations from experience.

Figure 5.3 Successful navigation via network-based wall following

6 Conclusion and Future Work

This research compared the performances between a conventional bug algorithm approach and a network approach to indoor navigation. For the conventional bug algorithm, a controller was created to compute the parameters that enable the robot to successfully navigate in a complex environment based on the sensation of walls and other obstacles while keeping a safe distance from them. The deep learning approach taken here is a variation on the bug wall-following technique. However, instead of composing an explicit algorithm that interprets depth-camera data, a Convolutional Neural Network was trained to interpret camera data and perform navigation. A motivation for this is that it can be very difficult to compose explicit code that correctly interprets depth-camera information in terms of what is and what is not navigable for a non-circular robot. In contrast to writing explicit code for such sensory interpretation, the CNN is trained based on examples. Specifically, stored images from dynamic simulation were acquired and labelled as "safe" or "unsafe" based on whether a collision did occur a short time later. These examples provided associations that were found to be true, whether or not the programmer understood how to interpret the 3D images.

The labelled data was automatically generated for 8 days straight. More than 90,000,000 images were collected for all the labelled data. The trained network achieved an accuracy of roughly 98%. The network was deployed in simulation, and the same network processed images from three cameras: one forward camera and a left-front and a left-rear camera. Images from these three cameras were interpreted simply as "safe" or "unsafe". An algorithm acted on these values to keep the robot following alongside walls and other obstacles, keeping the barriers to the left side of the robot. The algorithm was shown to be effective in complex environments.

In future work, many extensions and improvements could be performed. Notably, the present work did not complete implementation of the "bug 2" algorithm [41]. The premise of this algorithm is that a robot is to navigate from point A to point B using the equivalent of GPS and ranging sensor with no a priori information of its environment. To do so, the robot can construct a default path--a straight line in GPS coordinates. As the robot progresses, if it encounters an obstacle in its way, it can circulate about the perimeter of the obstacle until it rejoins its original default path, as illustrated in Fig 6.1.

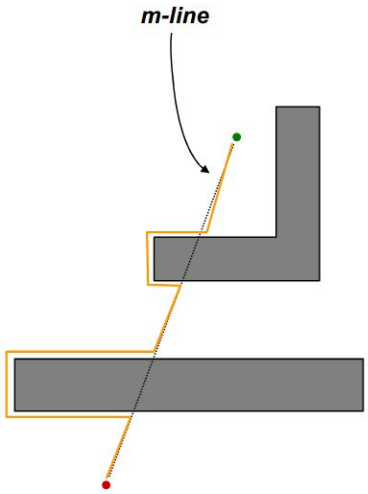


Fig 6.1: bug2 algorithm with M-line

Including the M-line in the present work could be accomplished by providing virtual "safe" and "unsafe" signals that correspond to treating the M-line as a virtual barrier. With this addition, the present navigation algorithm would inherit the maze-solving guarantees of bug algorithms.

In future work, alternative network architectures could be considered as well. For example, Long Short-Term Memory might be useful in interpreting sequences of images to achieve higher reliability. An important extension would be to test the present results in a physical implementation. Prior work has shown that training based on synthetic images can perform poorly when operating on real sensory data. However, it has also been shown that training data from physical sensors can be augmented with virtual data to achieve good performance in practice while dramatically reducing the amount of physical training data required. This should be explored in the present context.

Another improvement would be simplification of the integration of ROS and TensorFlow. Last year TensorFlow created a library in ROS. This feature can be used in ROS Kinect where the convolutional network can be invoked and trained in C++ code with the built-in TensorFlow library.

```cpp
#include "tensorflow_ros_test/lib.h"

// Here you can include any TF headers you want.

#include "tensorflow/core/public/session.h"
#include "tensorflow/core/platform/env.h"

using namespace tensorflow;

int do_tensorflow(const char *model_path, float *result) {
    // Initialize a tensorflow session
    Session* session;
    tensorflow::SessionOptions options = SessionOptions();
    options.config.mutable_gpu_options()->set_allow_growth(true);
    Status status = NewSession(options, &session);
    if (!status.ok()) {
        std::cout << status.ToString() << "\n";
        return 1;
```

Figure 6.2 sample code of the integrated TensorFlow function in ROS [26]

The write up remains the same style, but this integration has greatly reduced computation time from using TCP portal for network and ROS bridge.

Finally, as noted earlier, it may be desirable or necessary to modify the steering controller to operate more smoothly than switching based on safe/unsafe sensor interpretations. The network controller was based on two analogue outputs from the network: 'safe' and 'unsafe'. In a sense, 'safe' indicates that the sensor image predicts there will be no collision within 0.15 meters, and 'unsafe' indicates a prediction that there will be a collision within 0.15 meters of forward travel. However, this information does not provide any context of how far the input image is. As a future work, if the network could give three outputs 'far', 'medium' and 'near', then the robot could better adjust its behavior when following the wall. 'Near' can be defined as following the wall or getting too close to the wall. 'Medium' can be defined as getting away from the wall but still following the wall. 'Far' can be defined as either the wall disappearing on the left or the robot is encountering a sharp turning corner. The network setup would be more

complex with the ability of giving three outputs, but this could improve the performance of the navigation.

7 Bibliography

[1] Y. Li, J. Wang, T. Xing, T. Liu, C. Li and K. Su, "TAD16K: An enhanced benchmark for autonomous driving," 2017 IEEE International Conference on Image Processing (ICIP), Beijing, 2017, pp. 2344-2348.URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8296701&isnumber=8296222

[2] G. M. Gandhi and Salvi, "Artificial Intelligence Integrated Blockchain For Training Autonomous Cars," 2019 Fifth International Conference on Science Technology Engineering and Mathematics (ICONSTEM), Chennai, India, 2019, pp. 157-161.URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8918795&isnumber=8918696

[3] P. van Turennout, G. Honderd and L. J. van Schelven, "Wall-following control of a mobile robot," Proceedings 1992 IEEE International Conference on Robotics and Automation, Nice, France, 1992, pp. 280-285 vol.1.URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=220250&isnumber=5764

[4] “Wall Following programming” [online.] Available: http://students.iitk.ac.in/robocon/docs/lib/exe/fetch.php?media=robocon16:programming:wallfollowing.pdf [Accessed: 21-Jan-2020].

[5] W. Cui, C. Wu, Y. Zhang, B. Li, and W. Fu, “Indoor robot localization based on multidimensional scaling,” *International Journal of Distributed Sensor Networks*, vol. 11, no. 8, Article ID 719658, 2015. View at: Publisher Site | Google Scholar

[6] W. Cui, C. Wu, W. Meng, B. Li, Y. Zhang, and L. Xie, “Dynamic multidimensional scaling algorithm for 3-d mobile localization,” *IEEE Transactions on Instrumentation*

and Measurement, vol. 65, no. 12, pp. 2853–2865, 2016. View at: Publisher Site | Google Scholar

[7] W. Cui, L. Zhang, B. Li et al., "Received signal strength based indoor positioning using a random vector functional link network," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 5, pp. 1846–1855, 2018. View at: Publisher Site | Google Scholar

[8] J. Fuentes-Pacheco, J. Ruiz-Ascencio, and J. M. Rendón-Mancha, "Visual simultaneous localization and mapping: a survey," *Artificial Intelligence Review*, vol. 43, no. 1, pp. 55–81, 2015. View at: Publisher Site | Google Scholar

[9] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: a new learning scheme of feedforward neural networks," in *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, Budapest, Hungary, July 2004. View at: Publisher Site | Google Scholar

[10] H. Zou, H. Jiang, X. Lu, and L. Xie, "An online sequential extreme learning machine approach to WiFi based indoor positioning," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pp. 111–116, Seoul, South Korea, March 2014. View at: Publisher Site | Google Scholar

[11] T. Tongloy, S. Chuwongin, K. Jaksukam, C. Chousangsuntorn and S. Boonsang, "Asynchronous deep reinforcement learning for the mobile robot navigation with supervised auxiliary tasks," 2017 2nd International Conference on Robotics and Automation Engineering (ICRAE), Shanghai, 2017, pp. 68-72.URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8291355&isnumber=8291337

[12] L. Bill and H. Shahnasser, "Development and Implementation of an Autonomously Driven Vehicle Prototype," *2019 IEEE 2nd International Conference on Electronics Technology (ICET)*, Chengdu, China, 2019, pp. 310-314.URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8839507&isnumber=8839311

[13] Y. Chang, P. Chung and H. Lin, "Deep learning for object identification in ROS-based mobile robots," *2018 IEEE International Conference on Applied System Invention (ICASI)*, Chiba, 2018, pp. 66-69. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8394348&isnumber=8394245

[14] Z.Chen, "Deep-Learning Approaches to Object Recognition from 3D Data," Case Western Reserve University, August 2017.

[15] Young, Chih-Chieh & Liu, W.-C & Hsieh, W.-L. (2015). Predicting the Water Level Fluctuation in an Alpine Lake Using Physically Based, Artificial Neural Network, and Time Series Forecasting Models. Mathematical Problems in Engineering. 2015. 10.1155/2015/708204.

[16] P. J. Werbos, "Backpropagation through time: what it does and how to do it," in Proceedings of the IEEE, vol. 78, no. 10, pp. 1550-1560, Oct. 1990.

[17] - , "Generalization of backpropagation with application to a recurrent gas market model," Neural Networks, Oct. 1988.

[18] Zhao Yanling, Deng Bimin and Wang Zhanrong, "Analysis and study of perceptrons to solve XOR problem," The 2nd International Workshop on Autonomous Decentralized System, 2002., Beijing, China, 2002, pp. 168-173.

[19] L. Jacobson, "Introduction to Artificial Neural Networks - Part1." [Online].Available: http://www.theprojectspot.com/tutorial-post/introduction-to-artificial-neural-networks-part-1/7. [Accessed: 10-Jan-2017].

[20] T.M.Mitchell, Machine Learning. McGraw-Hill, 1997.

[21] Alex Smola and S.V.N. Vishwanathan, Introduction to Machine Learning. Cambridge University Press, 2008.

[22] YouTube. "Final NN pub des." Online video clip. YouTube, 17 April 2020. Retrieved from https://youtu.be/HlDHVZchF7A

[23] Lentin Joseph, ROS Robotics Project. Packt, March 2017.

[24] T.J Pech, "A Deep Learning Approach of Evaluating the Navigability of Off-road terrain from 3-D imaging", Case Western Reserve University, August, 2017

[25] W. Newman, A Systematic Approach to Learning Robot Programming with ROS. Taylor & Francis Group, 2018.

[26] https://github.com/tradr-project/tensorflow_ros_test/blob/kinetic-devel/src/lib.cpp

[27] V. Mazzari, "Robotics simulation scenarios with Gazebo and ROS." [online]. Avaliable:https://www.generationrobots.com/blog/en/robotic-simulation-scenarios-with-gazebo-and-ros/.[Accessed: 21-Jan-2020].

[28] "ROS-Data display with Rviz." [online]. Available: https://www.stereolabs.com/docs/ros/rviz/. [Accessed: 21-Jan-2020].

[29] "Make a model." [online]. Available: http://gazebosim.org/tutorials?tut=build_model. [Accessed: 23-Jan-2020].

[30] "Adding Physical and Collision Properties to a URDF Model." [online]. Available: http://wiki.ros.org/urdf/Tutorials/Adding%20Physical%20and%20Collision%20Properties%20to%20a%20URDF%20Model. [Accessed: 23-Jan-2020].

[31] "Blender." [online]. Available: https://www.blender.org/. [Accessed: 23-Jan-2020].

[32] "Advantages of Simulation." [online]. Available: http://web.cs.mun.ca/~donald/msc/node6.html. [Accessed: 21-Jan-2020].

[33] K.Bayer, "Wall Following for Autonomous Navigation", Columbia University, Summer 2012.

[34] "History of Machine Learning." [online]. Available: https://www.doc.ic.ac.uk/~jce317/history-machine-learning.html. [Accessed: 22-Aug-2019].

[35] M.Nielsen, Neural Networks and Deep Learning. Lambda, Dec 2019.

[36] "TCPROS." [online]. Available:http://wiki.ros.org/ROS/TCPROS. [Accessed: 21-March-2019].

[37] "Services." [online]. Available:http://wiki.ros.org/Services. [Accessed: 21-March-2019].

[38] "Multipath TCP Deployment." [online]. Available: https://www.ietfjournal.org/multipath-tcp-deployments/. [Accessed: 22-March-2019].

[39] "Navigation.RViz." [online]. Available: http://4.bp.blogspot.com/-oQo8001Tow4/VasE4Eahk0I/AAAAAAAA3ZM/_OZsb6JjRPc/s1600/fetch_rviz_gz_costmap.png. [Accessed: 15-April-2020].

[40] "Wall Follower."[online]. Available: https://www.philohome.com/wallfollower/wallfollower.htm. [Accessed: 13-Jan-2020].

[41] "Robotic Motion Planning Bug Algorithms." [online]. Available: https://www.cs.cmu.edu/~motionplanning/lecture/Chap2-Bug-Alg_howie.pdf. [Accessed: 13-April-2020].

[42] YouTube. "Automated data collection in RViz view." Online video clip. YouTube, 21 April 2020. Web. 21 April 2020. Retrieved from https://youtu.be/E1ewdl5fsfo

[43] YouTube. "Wall following controller." Online video clip. YouTube, 12 Feb 2020. Web. 12 Feb 2020. Retrieved from https://www.youtube.com/watch?v=VwU-HM3QN_8

[44] YouTube. "Complex en controller." Online video clip. YouTube, 27 Mar 2020. Web. 27 Mar 2020. Retrieved from https://www.youtube.com/watch?v=UmRiBZET7NE

[45] YouTube. "Final NN pub des." Online video clip. YouTube, 17 April 2020. Web. 17 April 2020. Retrieved from https://www.youtube.com/watch?v=HlDHVZchF7A&t=91s

[46] X. Wei, E. Dong, C. Liu, G. Han and J. Yang, "A wall-following algorithm based on dynamic virtual walls for mobile robots navigation," *2017 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, Okinawa, 2017, pp. 46-51.doi: 10.1109/RCAR.2017.8311834 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8311834&isnumber=8311819

[47] YouTube. "Wall following." Online video clip. YouTube, 25 Jan 2020. Web. 25 Jan 2020. Retrieved from https://www.youtube.com/watch?v=lMx27Iov9qs&t=1s

[48] "Autodesk FUSION 360." [online]. Avialble:https://www.autodesk.com/products/fusion-360/overview?mktvar002=3515888|

SEM|1618310824|96814085015|kwd-52459804111&gclsrc=aw.ds&&ef_id=CjwKCAjwnIr1BRAWEiwA6GpwNVQv24Al7swMeWJIlMwqfTt91DFmJl4fh6XZe5dl2W_k80PH8CRVjRoCmGgQAvD_BwE:G:s&s_kwcid=AL!11172!3!420771664709!b!!g!!%2Bfusion360!1618310824!96814085015&mkwid=ssC8dsv0o|pcrid|420771664709|pkw|%2Bfusion360|pmt|b|pdv|c|slid||pgrid|96814085015|ptaid|kwd-52459804111|pid|&utm_medium=cpc&utm_source=google&utm_campaign=GGL_DM_Fusion-360_AMER_US_eComm_SEM_BR_NEW_BMM_ADSK_3515888_&utm_term=%2Bfusion360&utm_content=ssC8dsv0o|pcrid|420771664709|pkw|%2Bfusion360|pmt|b|pdv|c|slid||pgrid|96814085015|ptaid|kwd-52459804111|&gclid=CjwKCAjwnIr1BRAWEiwA6GpwNVQv24Al7swMeWJIlMwqfTt91DFmJl4fh6XZe5dl2W_k80PH8CRVjRoCmGgQAvD_BwE [Accessed: 24-April-2020].

[49] YouTube. “Network controller with left side wall following.” Online video clip. YouTube, 5 May 2020. Web. 5 May 2020. Retrieved from https://www.youtube.com/watch?v=49pwdXmRcEk

[50] “Learning rule demonstration.” [online] Available: https://lucidar.me/en/neural-networks/learning-rule-demonstration/ [Accessed: 21-March-2019].